A METHODOLOGY TO CONFIGURE A MINIMAL, APPLICATION-SPECIFIC OPERATING SYSTEM FOR DISTRIBUTION AND OPERATION

BY

ZACHARY JAMES YORDY

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Professor David Nicol

ABSTRACT

NetAPT is a tool that helps enterprise and utility customers validate their network security policy. While the tool gives helpful results, it relies on lots of external programs to help it perform its powerful analysis. These dependencies often take hours or days to correctly install and configure. Users need a solution that helps them run the tool quickly, a distribution in which this deep configuration has already been performed. This thesis details how to create a minimal, application-specific ISO of a Linux operating system that can be booted live, installed to a computer, or virtualized across many different platforms.

The author first looked at Linux From Scratch as a way to create a completely minimal and bloat-free system. Once the system had been compiled, it failed to run properly and was not small enough. The author next looked at customizing the Ubuntu installer, which well supported the application and its dependencies.

The final distribution produced a small, branded ISO that could be run as a LiveCD or installed as a traditional operating system. This makes it easy for users to download a copy of the tool and use it right away without the need for any configuration.

To Aspen and to my parents, for their years of love and support

ACKNOWLEDGMENTS

These accomplishments would not have been possible without the support of many people.

The author would first like to thank David Nicol for the opportunity to work with him, and for his guidance over the past two years.

The author's research team was an additional area of support. Efforts from David, Mouna, Edmond, Robin, Rakesh, George, Jenny, Bill, and Kate do not go unnoticed. This would not have been possible without their hard work and dedication.

Next, the author would like to thank his fiancée, Aspen, for putting up with his bad moods, stress, and complaining. It is hoped this was a worthwhile undertaking.

Additionally, the author's network of family and friends has been critical, as they have given him an immeasurable amount of encouragement and support. These relationships have been imperative to the author's motivation and endurance during this long process. Jake, Mike, and Aspen were extremely constructive and willing helpers.

Finally, the author appreciates and recognizes the Information Trust Institute and the University of Illinois, for their amazing financial support and endeavors to make this experience possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

NetAPT	Network Access Policy Tool
RVM	Ruby Version Manager
RPC	Remote Procedure Call
OS	Operating System
GUI	Graphical User Interface
CD	Compact Disc
DVD	Digital Video Disc
LFS	Linux From Scratch
DSL	Damn Small Linux
RAM	Random Access Memory
URL	Uniform Resource Locator
UID	User ID
GID	Group ID
IP	Internet Protocol

CHAPTER 1

INTRODUCTION

NetAPT, or *Net*work *A*ccess *P*olicy *T*ool, is an application created and developed by the University of Illinois that fulfills a very specific but useful purpose. Concisely, NetAPT parses firewall configuration files and draws an interactive map to help validate network security policy. Although the tool usefully assists enterprise and utility customers in managing the security devices on their network, it has a downside.

In order to make use of the impressive technology that makes NetAPT and its powerful analysis engine possible, the tool takes advantage of external libraries and programs. While these help NetAPT to do some of the heavy lifting, successful and correct installation of all its dependencies is much more complicated than it should be. The process of installing all the dependencies required to run NetAPT often takes several hours, even for a seasoned user. For a new user adjusting to a fresh learning curve, this initial system configuration can take multiple days. NetAPT installers for Windows and Mac try to help ease the configuration of these dependencies, but installation issues can manifest themselves in unexpected ways, even when the user thinks everything has been configured correctly.

Clear need exists for a solution that enables new users to be running NetAPT straightaway. Users want to get real results into their hands quickly, and the tool also takes time to perform its analysis. Historically, a cross-platform installer has been used, though restrictions on different operating systems make for an overly fragmented and inconsistent installation process. In short, such a frustrating experience details the need for a solution that provides consistency across many platforms. This thesis details how to create a minimal, application-specific ISO of a Linux operating system that can be booted live, installed to a computer, or virtualized across many different platforms.

1.1 Background

Firewalls are incredibly functional devices used in just about every computer network. Firewalls check and scan all network traffic en route to ensure that each network packet reaches its intended destination. More importantly, they deny access by blocking any traffic not meant for a particular destination. If cars on a road represent packets traversing a network, a firewall might best compare to a road block that only lets in local or construction traffic. Firewalls ensure that network traffic only goes where it is intended. For example, within a banking network, one might want to ensure that a very limited set of computers have access to the servers containing sensitive financial data.

Firewalls use a special file that dictates precisely how they should work, aptly named a *firewall configuration file*. These files work by specifying rules that describe exactly which computers are allowed to communicate within a network. The `allow` rule permits two computers to talk to each other; similarly, one can specify that two computers should not communicate with a `deny` rule. Firewall configuration files have a large degree of flexibility in that `allow` and `deny` rules can be specified for a single host, a subnetwork, or even a group of hosts. A configuration file can also specify in which direction (inbound or outbound) those connections take place. For example, a configuration file might mandate that host `192.168.0.15` only send messages to `192.168.2.76`, while disallowing messages in the other direction.

Although such explicit rules enable a granular degree of control, firewall configuration files often get very long. Connections allowed and denied on such a detailed basis mean that a standard firewall configuration file might contain hundreds or even thousands of rules. Since each and every rule needs to accurately specify permissible connections, auditing a configuration file becomes particularly difficult. The fact that most networks use multiple firewalls only further complicates the problem.

With multiple firewalls, the task of ensuring correct configuration of each firewall becomes nearly impossible. Not only does one have to consider each and every rule on every single firewall, but also how those rules might be affected by or interact with other rules on every other firewall. If each firewall contains hundreds of rules, this sort of policy validation grows exponentially to millions of different possibilities. NetAPT, developed to ensure that firewalls have been configured correctly, precisely fills this niche. The tool helps

to check the specified guidelines about device interaction on a network. For more information about exactly how NetAPT works, see Appendix A.

NetAPT displays every single flow allowed through the network on a clickable, draggable, interactive map (Figure 1.1). When a user clicks on a node, all possible incoming and outgoing connections to or from that node are highlighted. For example, the user can find their critical network and determine exactly which other machines on the network have access to important information. If a connection looks suspect or needs to be changed, the tool will inform the user exactly what rule or rules caused that connection to be allowed. NetAPT informs about specific rules within specific configuration files, enabling the user to change a rule and see the effect of that change with another analysis.

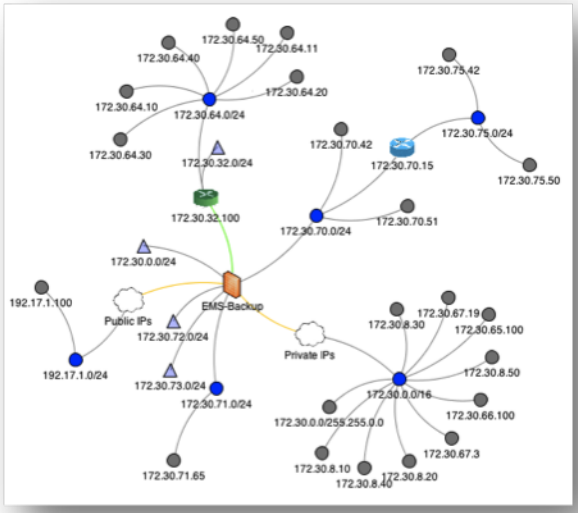


Figure 1.1: NetAPT's Network View

1.2 The Need for Easy Configuration

NetAPT has become very valuable to enterprise customers and utilities who use it to validate their network security policy. NetAPT, while powerful, relies on lots of other tools to help it perform its analysis. It takes advantage of The OpenSSL Project's *OpenSSL* to help with cryptographic functions

and processing, Apache's *Xerces-C++* for parsing and maintaining representations of state with XML, and a passive network mapping application called *ANTFARM* to form a visual representation of the network. When first installing NetAPT on a user's computer, all of these programs have to be installed as well.

Of these dependencies, OpenSSL and Xerces-C configure effortlessly. Standard `./configure`, `make`, and `make install` commands quickly set up the downloaded source packages. Configuring the rest of NetAPT's dependencies proves to be an entirely different experience, however. ANTFARM, bundled as a *Ruby gem* (a little package that adds functionality onto Ruby itself), necessarily requires Ruby to be installed. ANTFARM also relies on several additional gems (including *Ruby on Rails*) for proper execution. Since some of NetAPT's parsers also run on Ruby, installation is all but required, however Ruby itself makes for very hard configuration and installation.

It is fair to describe Ruby as fickle: as a program, it is quite particular about the directories where libraries and binaries are installed, and one library or binary in the wrong place may require complete reinstallation. Ruby, similar to Python, currently ships in two fairly different versions (`1.9.3` and `2.0.0`), and different products may depend on either version in order to work. Unlike Python, multiple versions of Ruby on one system make it extremely temperamental. Installation and configuration of Ruby is so difficult, in fact, that a product called RVM (Ruby Version Manager) has been developed in order to streamline the process and make it just a little less difficult. ANTFARM requires more than just Ruby, though.

ANTFARM keeps track of the visualization of network nodes in a MySQL database. Not only does ANTFARM need MySQL, but it also needs the MySQL gem, developed for compatibility between MySQL and Ruby. Although these dependencies install more easily than the programs that require them, they still make for an extra step before the tool can run properly. Finally, ANTFARM has very specific prerequisites. The program, which stores all of its information in a special `.antfarm` folder in the user's home directory, simply will not work if it lacks read or write access or runs as the wrong user.

ANTFARM has been developed on a much smaller scale than things like Ruby or MySQL. In its prime a community of a few members regularly contributed to the project. Now maintained by only one lead developer,

ANTFARM is no longer in active development [1]. For this reason, it lacks the extensive documentation and online support that might help users solve problems with more popular products. ANTFARM also has somewhat poor documentation and only a small degree of error output, making modern use even harder for those who encounter problems.

All of these dependencies and configuration hiccups create difficulty rather than enabling a user to quickly be ready to run the tool. The system cannot be configured for use in less than a few hours, and users often take a full day or even several days to complete configuration of NetAPT and all of its dependencies.

Configuration problems frequently manifest themselves in strange ways with NetAPT. Even though the user might be under the impression that the whole installation completed successfully, the engine (the heart of the analysis) might not run, or might error out. Installers have been developed for Windows and Mac, but they require certain programs to be installed already, and do not work as well as an installer should. No installer exists for Linux.

There is a definite need for an easier way to run NetAPT. Its myriad of dependencies make it extremely hard to correctly configure and install. Even though NetAPT's results help enterprise users, they want an easier way to use the tool. This thesis describes a way to provide an ISO of a Linux distribution to a customer that can easily be run as a LiveCD or installed to a computer. The main advantage of this technique lies in the fact that setup takes only as long as the time to download the ISO. The ISO comes with all required dependencies already installed; users, directories, libraries, and binaries are already configured. This system has flexibility since the Linux distribution can be run on any machine as a LiveCD, easily installed to a machine, or run as a virtual machine.

Linux was chosen as the host operating system (OS) because it has many advantages over Mac OS and Windows. First, a standard Linux installation has a much smaller footprint than that of a standard Mac or Windows installation. A full-featured Linux installation uses about 4 GB of hard drive space when completely installed, while Mac and Windows average closer to 8 GB or more. Second, Linux is the most customizable of the three major OS choices. It can be stripped down to include exactly those tools and libraries that are needed, and nothing else. Linux has the option to run either from

the command line entirely, or through a very lightweight Graphical User Interface (GUI). It runs as a bootable CD very easily, and almost every kind of hardware supports it. Third, many different *flavors* of Linux operating systems exist. Some distributions choose to sacrifice power for extremely small size, while others are so full-featured that they have just about everything the average user might need. Linux distributions offer the ability to choose the most convenient distribution that fits the given operating system constraints. Due to the sheer number of flavors and implementation differences in each, three qualities were identified as critical to the final distribution choice: a small footprint, the ability to run as a LiveCD, and a very small amount of bloat (if any at all).

According to Akamai's *State of the Internet*, the average Internet connection in the United States now weighs in at about 7.4 Mbps [2]. With this connection speed (and no overhead), it would take almost 37 minutes to download a 2 GB file. For this reason, the distribution needs to be as small as possible in order to minimize the time taken to get it into the hands of users. When a user downloads an ISO, it will come with a system that has already been preconfigured. Most importantly, a copy of the tool is already installed to this system, but no upgrade mechanism exists. This will require that users download a completely new system every time an update to the tool is released, which makes the need for a small system that much more pertinent. A system with a fairly small footprint would be about 1 GB, meaning that it would take just over 18 minutes to download each update on an average connection, much better than a bigger system where the download could easily take an hour.[1]

Most distributions come with the ability to try the system out before installation to a hard drive. This benefits the user as they can easily boot into the system to perform quick tasks without needing to take the time to install the full system. This feature, called a Live CD, often acts as a way to "test out" a system before deciding to use it, but such a component would be highly valuable for an implementation of NetAPT as well. A user of a system with such a feature could copy the ISO to a CD or DVD, boot into the system, and use it on any computer without affecting the underlying host operating system. For example, a user could take their work laptop

[1]Even a system that downloads in an hour saves time compared to the time taken to install and configure NetAPT's required dependencies.

with a Windows operating system and boot from a CD that would contain everything needed to run the tool, temporarily booting into a special Linux operating system, where NetAPT could be used as it was intended. Once NetAPT was no longer needed, the user could simply eject the CD and restart the computer. The computer would boot back into the Windows operating system, where everything would be found just as it was left, with all files and programs unchanged.

Finally, the operating system needs to contain very little bloat. With so many different distributions to choose from, each one comes with its own set of applications. This operating system is designed to be purpose-built to run NetAPT. Since the main function of the system lies in running the tool, extra utilities such as an office suite or media player should be eschewed. A small degree of useful extra utilities is acceptable; however, they might take up valuable space that would otherwise make for a faster download. Linux operating systems are fairly modular, and should allow for this degree of control over exactly what packages are included in the final distribution.

In order to fit the operating system into the smallest space possible, the author first looked at techniques to build an operating system that only had the tools and utilities that were directly needed. A project called Linux From Scratch (LFS) was used. This project provides extensive instructions on how to compile an operating system from the ground up [3]. Building the system this way puts together exactly those parts of an operating system that are needed while stripping away the superfluity. Although it would have been far too difficult to customize the kernel and application libraries to this specific application, a minimal operating system was built and configured. Unfortunately, the operating system failed to function as desired (or almost at all), and was somewhat bigger than the space requirement would allow. Consequently, a new tack was in order.

The new method, outlined in this thesis, ended up being much more successful than the first, and aims to create an application-specific operating system (ASOS) from a small general purpose operating system (GPOS). Rather than spending the time and resources necessary to create the smallest possible kernel and operating system, this method trades off development time for space and hardware requirements. A general purpose operating system is first used, but rather than take the time to pare down the system to only the exact application calls needed to run NetAPT, this method simply

builds on top of the GPOS. The end result is a product that takes up quite a bit more space, and also has hardware requirements that are significantly greater than a traditional ASOS. In return, the operating system has the flexibility to do just about anything desired, but also fulfills the application-specific need. The created operating system still fits well within the space requirements; to be sure, end users of the created product have much more relaxed space and hardware requirements than something like an embedded operating system might have.

While these instructions use NetAPT as a framework for all discussion related to creating this ISO, they could easily be used for any situation in which customization of any generic system is required. This thesis gives a framework for taking an existing installer and customizing it to fit the needs of a specific tool or system so that it can be packaged up into a small, distributable, bootable ISO.

CHAPTER 2

RELATED WORK

Even though NetAPT has not been used in a minimal, application-specific operating system, the technique of paring down an operating system to its required components is nothing new. Such an operating system structure was first proposed by Thomas Anderson in April of 1992 [4]. Even twenty years ago, he noted that costs for hardware were continuously becoming lower, a trend that continues today. At the time of Anderson's suggestion, people debated about whether the kernel of the operating system should be *monolithic*, including every operating system module that might be necessary, or whether it should be smaller and more nimble, pushing any operating system modules up the stack to more specific application-level servers [4].

Anderson instead proposed a new system, whereby the kernel would be as small and minimal as it possibly could be. It would support only those calls that were directly necessary to run a specific application. Furthermore, the core operating system would only be responsible for multiplexing requests for hardware resources and enforcing that the hardware was protected from applications. The rest of the operating system functionality, including things like managing hardware resources and inter-application communication, would be bundled as linked library routines within the applications themselves. This would provide the opportunity for modules to be included within the specific application that actually needed them, rather than being compiled into a larger monolithic kernel. Anderson provided several use cases for an operating system structure like this, many of which are still in use today [4].

The justification for a system like this makes sense. In fact, several instances exist in which an operating system only needs to perform one specific function. The primary goal of an operating system is to function as an interface between the applications that are executed on a machine and the hardware on which the OS runs. This provides an opportunity for multi-

ple applications to run on the same machine while transparently handling and metering requests for its hardware. The OS also ensures that its physical equipment is protected from an application trying to take advantage of, control, or otherwise overuse it. Operating systems become large because they need to support many different use cases by interfacing with a variety of different applications. Furthermore, although the operating system tries to make sure that no application gets unfair use of the available resources, these needs change depending on what application is running. This means that no operating system can fully cater to the needs of any one application. Although the abstraction layer of an OS is helpful and often necessary, the overhead of such an implementation means that applications are much slower interfacing through the operating system than running on bare hardware itself [5].

In 1995, only three years after Anderson's proposition, a survey revealed that at least nine major executions of application-specific operating systems were already in use. Each of these solutions had a different approach, and, naturally, its own advantages and disadvantages. Some of these implementations were more bare-bones, much like what was suggested by Thomas Anderson. Others chose to give the operating system a bit more control by letting it handle things like policy decisions or low-level communications protocols. One group even suggested a more modular architecture whereby the desired pieces could be included into the kernel and operating system, but any unnecessary modules would be left behind [5].

The four authors of the survey argued that each execution was somewhat hampered by the design decisions that were made, which resulted in a recommendation of their own more generalized solution to the problem. Instead of developing an operating system that only multiplexes requests for resources, they proposed an operating system that also handles threads and inter-process communication. This system, called a *microkernel*, bundles more than just a functionally minimal kernel and operating system. This helps strike a balance between outsourcing every possible extension of the OS and having a much more fully-featured kernel that supports every function that could possibly be needed. In their system, the authors recommend that applications be profiled either before being run or on their first run, so that the OS can best optimize its scheduling and resource handling to the application [5].

During this period in the late 1990s, most of the fledgling research about application-specific operating systems was focused around having an extremely small kernel and an operating system that only contained the most minimal functionality. This theory allowed applications to bundle any other necessary pieces of the operating system. This relates closely to the first technique tried in this thesis, where compiling a completely minimal operating system was attempted. However, major implementation differences existed. The methodology attempted in this thesis produced an operating system that had little bloat in terms of applications and user-facing features on the system, but did nothing to try to optimize (let alone minimize) the operating system or underlying kernel.

In the early 2000s, the approach of making an ASOS out of a minimal kernel and OS slowly began to morph. By 2001, at least sixteen additional major ASOS implementations had been created, with hundreds of other obscure executions. By this time technology had advanced significantly, as chips capable of running complete operating systems had become small and cheap enough to make them more ubiquitous. The discussion of application-specific operating systems began to switch to a natural utilization: embedded systems [6]. These systems, found on small, embedded chips in everything from digital watches to MRI machines [7], are often restrained in some aspect. These restraints necessitate a desire for a system that most efficiently uses the available resources, naturally suggesting a stripped down, application-specific embedded operating system.

Creating an ASOS to fit on an embedded system is anything but trivial. Often, embedded systems need to be low-cost. An embedded system, while performing a very specific function, usually only makes up a small part of the product in which it is embedded. Those in charge of specific costs for the whole project often do not oversee the development of the product from end-to-end. This places strict cost restrictions on the embedded system functionality of the product; the embedded system might also be constrained for physical size. Either way, this demand for low cost or small size is met by using underpowered processors with low available memory. Due to these extreme constraints, the system needs to perform its specific function with as little overhead as possible, or the system might be slow, unstable, or simply crash altogether. Such a minimal system can best be created by engineering every aspect of the system. This includes the kernel, available libraries, and

underlying operating system itself as well as any user-facing applications. Large investments of labor and money are usually required to ensure that the operating system as a whole takes up as few resources as possible at runtime.

This idea is associated with but separate from the goal of the main research methodology outlined in this thesis. The system detailed here is not a traditional application-specific operating system in that it is not intended to fit within the constraints typically required by conventional embedded systems, as outlined above. Rather, this thesis takes a broader definition of what it means to be an application-specific operating system: an operating system designed or modified to perform a particular function or run a specific user-facing application.

In the mid 2000s, new ideas ([8], [9]) started to shift the focus from developing and engineering application-specific embedded operating systems to a different technique. People started to carefully investigate creating application-specific operating systems by customizing GPOSes for the specific desired application. One survey of several different configurable operating systems by Jean-Charles Tournier argued that even though general purpose operating systems hamper performance, fixed operating systems are too restrained and do not provide enough functionality. Tournier noted that many general purpose operating systems have been modularized so that they are at least configurable. Even though they may ship with a wide range of functionality, they can be configured so that their functionality is narrowed down to the necessary part, which both decreases their size and improves their performance. The paper went on to list twelve different major, modern, configurable GPOS implementations, which shows that this specific problem can be solved in many ways [9].

A separate paper by Lamia Youseff started to look at ways that an ASOS could be automatically generated by a GPOS. She looked at two different methods for accomplishing this task. Virtual Machine Monitors, or VMM, is a framework that allows many different operating systems to share the same hardware, often called *virtualization*. Youseff's goal was to develop an application-specific operating system that can be virtualized and run on hardware with very little efficiency loss. Historically, computing systems have not been fast enough to be able to efficiently virtualize another completely separate operating system, but modern computing as well as new approaches

to the problem have breathed new life into the field. A more recent technique, called *para-virtualization*, virtualizes the hardware associated with a platform (with slight modifications) so that the virtualization is much more efficient and almost completely eliminates overhead. One execution of this technique reduced the overhead for computationally and I/O intensive applications from 5-8% and 56-88% with VMWare down to 0% and 8%, respectively. This is a significant reduction of overhead, and pushes the boundaries of what is possible with a virtualized system [8].

The second technique has been seen before, though not in this context. The method uses profiling and optimization to turn a GPOS into an ASOS. The application that is intended to be run on the operating system is *profiled*, or monitored, as it runs and executes code. The profiler is able to determine exactly what application and remote procedure calls (RPCs) are being made to the kernel, and can determine precisely what functions within the operating system are being utilized. The unnecessary parts of the operating system are then automatically stripped out, leaving an ASOS with exactly those libraries and necessary application calls and none of the extra bloat [10]. One popular implementation of a system like this uses a call-graph approach to figure out exactly what procedures are being called. Once the most utilized parts of the operating system are determined, the rest of the operating system (which is either redundant or contributes nothing new to the application) can be removed. The authors note that even though developing an ASOS by hand is slightly more efficient in terms of space, this automated technique takes much less time [8].

In broad strokes, these general sorts of solutions to create an ASOS are more in line with what is suggested in this thesis. The best solution outlined later does involve extending a general purpose operating system so that it performs the intended application-specific function. Even still, some subtle differences remain between the new trend above and what has been accomplished here. Research focusing on automatic generation of application-specific operating systems looks for a minimal end result. Even though the technique of starting with a GPOS differentiates from building a minimal embedded system, the goal to create a system that contains nothing more than the essential kernel and OS components to make the application work remains. This thesis does not require or strive for a completely minimal system. Rather, the aim is to create an operating system that has a small

footprint in terms of size, but such a constraint is not nearly as stringent as to require a functionally minimal system. The technique that produced the operating system with the smallest footprint actually had greater functionality than the minimal build that was attempted.

Some implementations of branded operating systems are very similar to the work that has been outlined here. The Education Operating Systems (EOS) Project put together a writeup of an OS that is a completely customized operating system that has been based on Linux. Like the methods outlined here, the EOS developers looked at two ways to create a new, branded distribution. The first involved building their own using LFS, while the second used an existing distribution like Ubuntu as a starting point for customization. Unlike the author, they had the foresight to forgo LFS as the best option because it would require lots of time and advanced technical OS skills [11].

The goal of the EOS project was to create a free, customized operating system that contained a set of software that might be useful to students. Like this thesis, the group who created the EOS used Ubuntu as an initial GPOS from which to build their product. Much like Edubuntu, another (Ubuntu-backed and sanctioned) education-based operating system, EOS was geared specifically towards users in that sector. The end result was an operating system that subsumed Ubuntu (and Edubuntu), including functionality above and beyond that included in Linux's most popular distribution. EOS managed to add useful features, including screenshot capture software, a diagram editor, a dock, and a new theme, among many other things [11].

While the EOS Project did end up with a similar end result to that of this thesis, some distinct differences are present as well. Although the end product was a customized, branded OS experience, the goals of the project were completely different. This thesis looks to provide a methodology that creates a customized, branded OS that can be run as a LiveCD or installed to a computer. Furthermore, the operating system has the requirement that it has to fit within a reasonably small space, so that each new iteration can be quickly downloaded. The EOS project had no such requirement, as it was meant to be installed locally. The resulting distribution was much larger, but also had more software available out-of-the-box. Finally, the EOS paper detailed generically the goals, solutions, and results of the project, but failed to mention how the operating system was actually customized. Based on

the amount of programs that were included as well as the extensive branding and customization options, it is likely that the authors of this project used a much different process for creating their operating system [11].

CHAPTER 3

EXPERIMENTAL DESIGN AND LINUX FROM SCRATCH

3.1 Choosing an Operating System and Distribution

Before creation of the minimal operating system could be attempted, there were many design decisions to be made—even though the field had been narrowed down to Linux, thousands of different Linux distributions remained, each with different advantages, disadvantages, and feature sets. Although certain distributions are moderately similar, some flavors boast stark differences. Ubuntu, Ubuntu Mini Remix, Ubuntu Net Install, Lubuntu, Bodhi Linux, Slim Pup, Damn Small Linux, and Linux from Scratch were all considered as initial options.

3.1.1 Ubuntu

Ubuntu is by far the most popular of all Linux distributions. Because of its popularity, Ubuntu has many tools available that make it easy to customize the system and create new distributions. It was designed to be user-friendly, but still has the ability to be completely customizable for those who consider themselves power users. It comes standard with all the tools that would be expected of any modern operating system, including popular Internet browsers, an open source office suite, media players, compilation utilities, and a great package manager [12].

Although Ubuntu has all the necessary tools of any full-featured operating system, it comes with lots of programs that would not be necessary for a specialized distribution like this. Ubuntu weighs in at over 4 GB fully installed, much bigger than allowable for this project. Easy tools help take a total snapshot of the system and package it up for distribution, though they produce an image that is too large to reliably download over the Internet.

Ubuntu would be great for those who need a full-featured OS to complement their tool. It would also necessarily be for those who do not have a very stringent space requirement. For those creating an operating system purpose-built to run one tool, Ubuntu might not be the best choice, unless many other consumer-level utilities of the OS are needed.

3.1.2 Ubuntu Net Install

While Ubuntu Net Install can be used to install Ubuntu, it is quite a bit different than the traditional installation media. Net Install comes in an amazingly small minimal CD between 5 and 30 MB (depending on system architecture) that downloads and installs any required packages as the installation of the system is taking place. Unlike regular Ubuntu installation media, which already contains versions of the most popular utilities, Net Install downloads the most recent, stable versions on-the-fly, obviating the need for system updates immediately after the system has been installed. Ubuntu similarly has an option to update system software as it is installed, but certain aspects of the system will still need to be updated immediately after the installation has completed [13].

Since Net Install lacks any usable programs, trying out the system before installation is impossible. The Net Install ISO is meant only to be used as an installer. An Internet connection is also mandatory, since no installation can take place without first downloading the required packages from the Internet. A system under 30 MB is quite impressive, although knowing that the installer contains no programs makes this much less so. Other distributions have a similar footprint, but also contain complete versions of many necessary utilities.

It is said that the user can control exactly which packages are configured upon installation of the operating system. Although this is most definitely a desired feature of a potential operating system, it does not appear that these packages can be explicitly chosen in any way. In the end, Net Install was just another way to get a fully configured Ubuntu OS, and the size of the finished installation reflected that as well. There was little size difference between either of the fully installed Ubuntu systems, although other versions of Ubuntu do not take up quite as much space.

3.1.3 Ubuntu Mini Remix

Ubuntu Mini Remix comes as a Live CD that has been stripped down to include only those utilities necessary to make the core operating system run. It comes as a fairly small ISO, fitting into a Live CD of about 200 MB. Although the distribution is not endorsed or sponsored by Canonical, a new version is created in lockstep with new Ubuntu releases. The distribution contains nothing more than core tools. It does not have any utilities for compiling new software, let alone any applications, or even a graphical user interface. Ultimately, the operating system itself cannot be easily installed or maintained; one can only make new releases using tools that have been created specifically to remaster ISOs [14].

Ubuntu Mini Remix would be a great option for those that want to build on top of a *very* basic operating system as a foundation. This would be especially true for users that only need a command line version of the operating system. One could customize the ISO to run their command line utility with ease. Ubuntu Mini Remix would not be a good choice for those that need their operating system to have a GUI. Compiling a window manager and graphical subsystem from scratch is not only a very complicated task, but tends to add 500 MB or more to the size of the installation media. At that point, the user would be better off with a more full-featured distribution of a similar size. Several Linux distributions still aim to fit their installation media into the size of a 700 MB writable CD, which would include much more utility for the same size.

3.1.4 Lubuntu, Bodhi, and Slim Pup

Lubuntu, Bodhi Linux, and Slim Pup are all flavors aimed at being minimal distributions. Lubuntu and Bodhi Linux are both based off Ubuntu, but use different window managers. Unfortunately, while these distributions are very lightweight, both of them focus more on using as few system resources as possible rather than having a small footprint. This makes them aptly suited for running on much older computers, but does nothing to help the problem of needing to have a small ISO [15], [16]. Lubuntu comes in only slightly smaller than Ubuntu, at around 3.6 GB installed [15]. Bodhi is smaller yet, but at 2.5 GB, is still too big to solve the problem of distribution [16]. Slim

Pup is somewhat different. It is a variant of Puppy Linux, a collection of distributions aiming to be lightweight and focused (somewhat) on ease of use. This variant of distributions are fairly full-featured, although they have many of the problems that hamper Lubuntu and Bodhi Linux. Puppy Linux and its variants are not based off of any specific distribution, which differentiates them a bit from most others. Regrettably, Puppy Linux and its variants suffer from fragmentation, lack of development (to a degree), and are not user friendly [17]. Although the three of these distributions are closer to a good candidate for this application, they still have some trade-offs that make them a poor choice for this project.

3.1.5 Damn Small Linux

Damn Small Linux (DSL) is a Linux-based operating system whose main goal is to fit in a (very) small size footprint. At first, it looked like this tool was the perfect one. The whole operating system as it now stands fits into a bootable Live CD of under 50 MB. In order to be so impressively small, a lot of unnecessary tools and bloat are stripped out from the system, which also includes utilities that can be compiled into a very small file size. To fit into its tiny size, the distribution has to make some sacrifices. It only runs on x86 processors, and only supports certain programs that can be compiled with a small footprint (specific window managers, text editors, or browsers, for example). Although it has an active community and has evolved into its own proper distribution over time, DSL has bigger problems [18].

First of all, although Damn Small Linux is still technically in development, the most recent stable version of the distribution was released in 2008. Secondly, DSL has its own package manager that can be used to install needed software [18]. Unfortunately, lots of tools that would be available for bigger distributions have not been compiled for this one. Since it runs such modified and specialized versions of software, any additional packages have to be compiled specifically to run on DSL. On another distribution, this would ordinarily be an easy problem to solve, but to help it fit into its small footprint, DSL does not offer any tools to compile software on the platform.

Damn Small Linux would be a great choice of operating system for those with extremely constrained space requirements. It manages to fit a very large

number of standard tools into an amazingly small space. It is not a good choice for those that can work with larger operating systems, or those that need much more configuration than standard utilities to run the operating system.

3.1.6 Linux From Scratch

LFS is much different than other distributions. As its name implies, Linux From Scratch is a distribution built and compiled entirely by the user, from nothing at all. Building a complete system from scratch is a large undertaking. One first has to decide on a core set of utilities that will be compiled, and then bootstrap those utilities to create the core operating system. Kernel headers have to be compiled, as do libraries to interface with the kernel, and of course the kernel itself. The end result is a system that contains exactly what is desired, and nothing else [3].

At first, it appeared that this method was not a good way to accomplish the goal at hand. Little was known about compiling many Linux utilities, much less a kernel and whole system to match. Designing a system like this, while enabling the compiler to have complete control over the end result, would be more difficult than taking an existing system and fitting it to perform the intended purpose. While compiling a completely new system from scratch would be a massive undertaking, this method started to become favored as it was considered in more detail. This distribution has many advantages.

Compiling one's own system from scratch enables complete control over the operating system and everything contained within it. The system is built using exactly those utilities that are needed, with none of the extra overhead of undesirable applications. Instead of shipping with a built-in GUI, window manager, office suite, and Internet browser, one can pick and choose exactly what to compile into the system. The user even controls whether lightweight applications, or their heavier, more fully-featured counterparts are desired. The beauty and simplicity of a system like this were compelling. While the process would most certainly take time to learn and reproduce, it would create a very simple and lightweight operating system that does exactly what is desired. Linux From Scratch was chosen as the distribution to package and run NetAPT. Unfortunately, what initially seemed like an elegant solution

would later create a larger set of issues that had to be overcome.

3.2 Building Linux From Scratch

To build one's own system from scratch requires quite a bit of work and dedication. Choosing a host system is the first requirement. Although it might not be immediately obvious, compiling a new Linux operating system necessitates starting with an existing Linux distribution as the *host*. This is the system in which the new OS is compiled, built, and packaged for use. The only requirements of the host system are that it provides a shell, a linker, and a compiler, and the ability to download files; however, it is often easiest to use a more common system like Ubuntu, Debian, or Red Hat that has a lot of features available right out of the box. Many of these systems have a *development* option that can be selected upon installation to load compilation tools onto the system. Once the host Linux system has been chosen and is ready, compilation can begin [3, p. 2].

3.2.1 Establishing the Compilation Environment

The first major step in building a Linux system is to set up the compilation environment. If desired, the new OS can be created on a separate partition of the hard drive. The filesystem can be initialized with the `mke2fs` command, and swap space can be designated with `mkswap`. Whether a separate partition is desired or not, a mount point for the new system has to be chosen. The LFS documentation recommends `/mnt/lfs` as the mount point for the system, denoted with the `$LFS` environment variable. It also recommends that all new packages be downloaded into a folder called "`sources`" [3, pp. 14-17], and that the first round of programs are compiled into a folder called "`tools`" [3, p. 26]. After this configuration has been performed, the system is ready to bootstrap a clean compilation environment into it.

The next step, compiling the *toolchain*, is possibly the most important in building a Linux system. The toolchain is a suite of basic development tools and utilities that make it possible to compile the rest of the new operating system from scratch. It is important that the new toolchain give a clean and isolated environment in which to continue development of the rest of the

operating system. This ensures that the new compiled OS is not introduced to subtle errors that might not be seen until the end of the build [3, p. 31]. Compiling an operating system from scratch is again a complex and delicate operation, and the number of things that can go wrong is impressive. To minimize the risk of errors being introduced to a new system, it is important to separate, or *sandbox*, the new environment as much as possible.

3.2.2 Compiling the Toolchain

The first tool compiled is `binutils`, which creates libraries that help with assembly, as well as static, dynamic, and hard linking on the system [3, p. 100]. The `configure` command of the next two tools, `GCC` and `Glibc`, performs important linker checks that are needed to determine whether correct installation has taken place [3, p. 31]. `Binutils` is compiled in two passes (as is `GCC`, compiled next). The first pass installs the core tool itself (the main package), while the second pass uses the first pass to build a good copy of these tools, now dynamically linked against the tools that were just built [3, p. 2].

Once `binutils` (pass 1) has been installed, the first pass of `GCC` is compiled. `GCC` contains the standard C and C++ compilers, and is needed for any further compilation on the system. Remember, `GCC` is being compiled from scratch because a clean system is desired. This first pass of `GCC` only installs the base compilers. These compilers are then used later to compile the rest of `GCC`. After this, the Linux API Headers are installed. The headers allow the C libraries, installed next, to interface with the kernel. When `Glibc` is configured, it installs the main C libraries that help allocate memory, peruse the file structure, read and write files, and interface with the kernel [3, pp. 37-41]. Finally, the second passes of both `binutils` and `GCC` are compiled, installing the full, clean versions of the tools, rather than the main core package. As these tools are installed, they are also dynamically linked against the sanitized toolchain.

The base of the toolchain has now been installed. The next four tools to compile (`Tcl`, `Expect`, `DejaGNU`, and `Check`) help with the various test suites that accompany `GCC` and `binutils`. Four separate utilities are absolutely essential to ensure correct operation of the core toolchain utilities. A few

more major and necessary tools and utilities round out the core toolchain. `Ncurses` is compiled to help with handling of characters in terminal screens (especially for the graphical menu configuration of the kernel). `Bash` is used as the shell, one of the most necessary programs of the whole build. `Bzip2`, `gzip`, `tar`, and `xz` are installed to help uncompress packages that are downloaded. Several collections of useful utilities are also included, such as `coreutils`, `diffutils`, and `findutils`. `Grep`, `patch`, and `sed` are incorporated to help change and search text. Finally, `make` is installed so that programs can actually be compiled [3, pp. 50-74]. A few other utilities are installed as dependencies or because they are later needed, but the programs detailed here make up the bulk of the toolchain.

A clean and sanitized toolchain has now been compiled from scratch into the `tools` directory. Compilation utilities, C and C++ compilers, kernel headers, C libraries, a shell, and several other useful programs form the toolchain with which the rest of the OS can be built. Before continuing, it is recommended that the `tools` directory be owned by the `root` user to avoid any configuration problems with users later on (`chown -R root:root $LFS/tools`) [3, p. 75]. The next step is to `chroot` into the `tools` directory to take advantage of the recently compiled toolchain. This change root command will enter a virtual environment and start a new shell that views the specified folder as the root folder of the system. Any changes that are made at that point would be identical to booting up into the `tools` directory and using the system contained therein. This clean environment will be bootstrapped to compile the actual desired operating system.

3.2.3 Compiling the Core System

Initial Preparation

Before any real OS compilation can start, a little preparation is necessary. Some initial device nodes need to be populated, and the virtual kernel filesystems need to be created, prepared, and mounted (specifically, the `/dev`, `/proc`, `/sys`, and `/dev/pts` directories) [3, p. 77]. After these small steps have been taken care of, the temporary tools system can be entered [3, p. 81]:

```
chroot "$LFS" /tools/bin/env -i \
```

```
HOME=/root                  \
TERM="$TERM"                \
PS1='\u:\w\$ '              \
PATH=/bin:/usr/bin:/sbin:/usr/sbin:/tools/bin \
/tools/bin/bash --login +h
```

This command changes root into the `$LFS/tools` directory. The switches of this command also specify the `root` user's home directory, what terminal to use, and the `PATH` environment variable. The system is now ready for complete configuration. Desired packages are compiled for inclusion in the new OS, and changes made at this point impact the operation of the compiled operating system. Remember, the clean utilities that have been compiled into the `tools` directory are now in use.

Compiling Necessary Tools Onto the System

Ordinarily, a new set of operating system utilities would now be compiled. This new set of programs would define exactly what was available to use in the operating system that was produced. Any desired tool could be added to this system; it is extremely modular. For a normal system, this would include recompiling every single one of the utilities listed above, as well as several other utilities that were not needed solely for compilation, including Internet configuration, man pages, system loggers, and a graphical terminal editor. Rather than create a full-featured system, it was desired to create a system that had as little bloat (and as small a footprint) as possible. Therefore, many of the suggested utilities would not be necessary.

This system is only intended to run one application, so compilation, compression, and standard system utilities, as well as user-facing applications (such as an Internet browser or office suite) and a terminal text editor are not needed. The system will still need the C libraries (`Glibc`) to perform basic functions, as well as the API Headers to interface with the kernel. These two tools take up well over 500 MB of space by themselves. It is quick work to configure and install these tools, however much more configuration is needed to create a whole system.

It was time to decide what other tools needed to be included into the system. `Bash` was an obvious choice, as it would be helpful to have a shell

available in which to execute commands. Even though it would take up another 100 MB, `coreutils` would also be useful, as the package includes common functions such as `chown`, `chroot`, `pwd`, `tail`, and `tee`, to name a few. `Shadow` and `sudo` were also installed to give the option to create and manage users, and execute commands as `root` [3, pp. 129-132]. At this point, the bulk of the core operating system was configured and installed. NetAPT's specific runtime dependencies had to be included next.

NetAPT has many dependencies that require quite a bit of extensive configuration. These utilities are enumerated and explained in much greater detail later. Although most of these were installed with little hassle, some required quite a bit more attention. Since NetAPT has a Java-based GUI, the OS needs a graphical user interface to be able to display the tool. Compiling a GUI into an operating system is anything but trivial. The operating system first needs a *window system environment*. Much like `Glibc`, the window system environment provides the libraries that interface with the kernel to support graphical user interfaces. The operating system needs kernel support to enable the GUI to run, as well as a *window manager* that executes and displays the actual user interface.

Compiling a GUI

Although there are other window system environments for Linux, `Xorg` (or just `X`) [19] is by far the most common. Used by almost every major distribution, `X` provides the backbone that makes GUIs like `GNOME`, `KDE`, and `LXDE` possible. `X` has recently changed its codebase to a modular system, so that, much like Linux itself, only the packages that are needed have to be installed. Unfortunately, installing and configuring this window system environment is laborious and complicated, possibly because `X` is the only available option [20].

The `X` window system environment needs a base of core packages that are common to any `X` installation. While it might make sense to combine these into one massive installation, these packages have to be downloaded and installed separately. Even worse, almost *300* separate packages make up this common core. To make this a little easier to handle, `X` is broken up into 15 different sections that need to be installed: the Protocol Headers, Xorg Utilities, `libXau`, `libXdmcp`, `xcb-proto`, `libxcb`, Xorg Libraries, `Xbitmaps`,

Xorg Applications, `xcursor-themes`, Xorg Fonts, `XKeyboardConfig`, `Luit`, `Xorg-Server`, and Xorg Drivers. While build scripts accomplish most of the heavy lifting, a few stubborn packages make the whole process difficult [20].

Take the Xorg Applications, for example. This package requires the Xorg Libraries as well as a program called `xbitmaps`, which requires Xorg Utilities, which in turn requires `pkg-config` and the Xorg Headers, most of which, luckily, have already been installed. It also requires `libpng`, which requires Mesa Lib, a much bigger package. Mesa Lib requires `expat` and `libdrm`, which in turn require `autoconf` and `automake`. It also needs `udev`, which won't work without `kmod` or `e2fsprogs`, which in turn needs `util-linux`. Finally, Mesa Lib recommends a program called `LLVM`, which, while not completely necessary, will help operation quite a bit. With many of these packages, special configuration instructions need to be applied [21].

`LLVM` should not have the `--enable-libffi` switch in the `configure` command, but needs the `LD_LIBRARY_PATH` environment variable set to `/usr/lib:/usr/local/lib:/tools/lib` to recognize python. `Expat` needs the `--with-expat=/tools` configuration switch, while `e2fsprogs` requires the `LDFLAGS` to be set to `"-L /tools/lib"`, which then should be unset after `udev` has been installed. After straightening out all of these dependencies, special configuration command-line switches, and environment variables, Mesa Lib (remember, a dependency for `libpng` and the Xorg Applications) failed to compile until the Xorg Libraries were recompiled with the correct `XORG_CONFIG` environment variable.

These steps were needed for just one of the 15 sections that make up the `Xorg` window system. After the main window system had been installed, the kernel needed to be compiled with support for a GUI. Drivers to use the system on certain hardware also needed to be compiled, along with a window manager (`fluxbox`) [22]. Installing the graphical user interface took far longer than expected because there were so many dependencies and exceptions to track down.

Admittedly, part of the difficulty in setting up this window system was out of desire for a minimal operating system. If a more complete OS had been installed as the base before compiling `X` was attempted, there surely would have been many fewer dependencies, requirements, and special configuration flags that needed to be dealt with, though a more fully featured system violates the requirement that this operating system have a small footprint.

3.2.4 Final System Configuration

After completely compiling and configuring the GUI, it was a relief to issue the `startx` command and see `fluxbox` spring to life. `Fluxbox` is a neat and flexible little window manager. It only takes up about 150 MB, which is minuscule compared to the 1.5 GB that heavier handed options like `GNOME` and `KDE` take up. With a single configuration file in the user's home directory, one can specify the desktop background, theme for the GUI, and even things like menu options on the left-click drop-down menu. Compared to more popular GUIs, `fluxbox` leaves the system with more available resources. It makes a nice compromise between size, ease of use, flexibility, and power compared to other super-lightweight window managers. It even supports Java applications, needed to run NetAPT.

The few other packages to be included with the main operating system were installed quickly and the system was soon ready for distribution. At around 1.5 GB, it was a bit larger than the target operating system size, though still within an acceptable range. The only major step left was to package the system into an ISO for distribution.

Packaging the ISO

An ISO is a disk image that typically contains all the files that might be present on a CD or DVD. The name ISO comes from the ISO (International Organization for Standardization) 9660 standard, which details the layout of the filesystem on a CD-ROM [23].[1] A packaged ISO is essentially the same as a CD-ROM, albeit in digital form rather than on a physical CD. Computers of all platforms (Windows, Mac, Linux) can read and utilize ISOs, making them a great way to package and distribute an operating system. Almost all virtualization solutions can boot directly from an ISO to run the operating system contained within it.

One of the most common utilities for packaging up an ISO is a tool called `ISOLINUX`. Part of the `SYSLINUX` project, `ISOLINUX` is specifically designed to make bootable ISOs from a directory. The user makes a fairly simple `isolinux.cfg` configuration file that specifies all the boot parameters, and

[1] The standard was actually proposed by ECMA (European Computer Manufacturers Association) as Standard 119, but was later adopted by the International Organization for Standardization.

then an ISO can be created using the `mkisofs` command [24]. This sort of process is widely used in mainstream distributions, including those such as Ubuntu.

The `ISOLINUX` configuration file can specify multiple boot options. Each boot option should have a menu label (seen by the user) and a kernel (from which to boot), and can also append any extra boot-time parameters that are desired, such as the root directory. One can specify which default boot option is desired, and when that option should boot, if no other option is selected [25].

Next, `ISOLINUX` necessitates changes to the kernel in certain situations. If an option is not configured or configured incorrectly, it may not be supported in the kernel and the system may not boot. Although kernel configuration is straightforward, this makes for another hurdle to cross.

`ISOLINUX` is very flexible, but is also somewhat difficult to configure. The myriad of options available within `ISOLINUX` are not only a feature but also possibly the tool's biggest downfall. The hardest part of making an ISO is the final configuration. If the options in the configuration file are not specified *exactly* right, the system will not boot. Since the ISO is a read-only medium, each iteration of the system (or even any small change to any file) requires regeneration of a whole new ISO. It took 9 different ISOs to get the configuration file just right. Although the bulk of the configuration file was correct, the ISO would not boot up until the exact root directory was appended.

After creating many slightly different ISOs, the system *finally* booted up and was nearly ready, which felt like great progress forward. Unfortunately, that feeling was short-lived. As the system booted up, error messages were presented about certain directories not being writable. The system *boot scripts* handle things like mounting directories for write access. The installed boot scripts had been created by the LFS project, and were fairly simple; they only mounted the necessary system directories as writable directories. The rest of the filesystem was read-only. Without write access to the filesystem, no changes could be made to files other than the 4 writable system directories. This meant that certain parts of the boot process failed to work, and, critically, the GUI would not start up. `X` needs write access to certain parts of the filesystem in order to modify files and create locks when the graphical user interface starts up. Without write access, there was no

GUI to be seen.

Creating a Ramdisk

Problems like this are often handled transparently in distributions by using what is called a *ramdisk*. A ramdisk is a file of a certain size consisting entirely of zeros. When the system gets booted up, the ramdisk gets loaded into system RAM and becomes usable as a temporary part of the system. If that portion of RAM is mounted by the filesystem, files within that directory become writable, although the changes do not carry over across system reboots. If the system had writable directories, it would behave exactly as desired.

Ramdisks have a few major pitfalls, however. First, since the ramdisk itself has to be zeroed out, the portion desired to be used in RAM has to be present on the ISO. For a few tens of megabytes, this is not a problem. If the system desires to use a ramdisk of many gigabytes, however, this directly affects the size of the distributed ISO. This distribution would need at least several hundred megabytes of ramdisk to be able to make the system writable as desired. The system was already at 1.5 GB, and did not have much room to grow while still fitting within the space requirements. After much research about ramdisks and trial and error, the system was still not writable.

3.2.5 Switching Gears

At this point, the project was now almost hopelessly behind schedule. It was starting to look like the system being built was not full-featured, and lacked infrastructure for certain critical function down the road. For example, users want to be able to run this system as a LiveCD and have the option to install it as well. If building a system from scratch even worked, it would only be bootable as a LiveCD. There would be no option to install it to a hard drive, and it would not have the flexibility of other distributions. There was one other major flaw with the operating system that had been compiled. The main use for NetAPT is to obtain information about how firewalls are configured and what the network looks like. Even though NetAPT performs this function well, the results take a long time to acquire. Because of this, NetAPT has been developed to be able to save the network topology and

analysis results so that any work performed does not have to be repeated. The easiest way to save this information on a Live system like this would be to use a flash drive, but the system did not have any mechanism for automounting flash drives. Although mounting a flash drive is probably technically possible, users would have to be familiar with the command line and the `mount` command to make any headway.

CHAPTER 4

CUSTOMIZING THE UBUNTU INSTALLER

After trying and failing to build a system from scratch, it was time for a complete change. There was clear demand for a system that would have the flexibility of a full-featured distribution, but would still be able to be significantly customized. The system that would ultimately be chosen would still need to have a small footprint and work as an installer or a LiveCD. In addition, the requirement of having as little bloat as possible was not completely necessary, especially if the system would indeed still fit into a small size. Having the system fit into a comparable size while packing in more features could actually be advantageous.

What limited the choice of fuller-featured systems before was that they often fit into a much larger footprint than what was hoped for. The installed size of a full-featured system is often close to 4 GB, and could be as much as 5 GB with everything that NetAPT required to run. A system with a similar degree of features and flexibility, but that fit into a much smaller package, was needed.

The solution was found in the Ubuntu installer. It was very small (around 700 MB), but still had the flexibility to do a wide variety of things, including support NetAPT [12].

When Ubuntu is distributed, it comes as a LiveCD and installer of around 700 MB. Most other major, more modern distributions have been pushed into LiveDVD territory, often needing 3 GB or more. In order to fit all that information into such a small package, the Ubuntu installer uses a compressed filesystem and downloads a lot of the extra utilities it might need as the installation is taking place. Most of the user-facing applications, like the GUI, internet browser, and office suite are compiled into the LiveCD, so they are available to use right away without needing to be downloaded [12]. Best of all, the Ubuntu LiveCD can be customized to include all the dependencies that NetAPT requires, meaning it can be configured to run NetAPT right

out of the box [26].

4.1 System Requirements

To compile the ISO itself, the host system needs to have 3 to 5 GB of available space, which helps to download and unpack the ISO and make customizations. It is recommended that the system have at least 512 MB of RAM and 1 GB of swap space to ease the task of compressing the filesystem and creating the ISO. The system needs to have `squashfs-tools` to be able to compress and uncompress the filesystem, and `genisoimage` (which includes `mkisofs`) to actually generate the ISO itself. The host system also needs to have an Ubuntu kernel with *squashfs* support so that the kernel can actually perform the compression. Squashfs support has been built into the kernel since Ubuntu version 6.06, so any recent version of Ubuntu will do. Finally, it is easiest to use a system that has the same architecture (`Amd64` or `i386`) as the ISO that it is desired to create. In short, it is easiest to use the exact same host system as the ISO will be. It is extremely hard to create an ISO intended for a different architecture [26]. For the Live NetAPT system, an i386 ISO provides maximum compatibility, so a stock Ubuntu i386 build will be used as the host system to complete the compilation. `Ubuntu 12.04 LTS` can be used since support for the system will be extended into 2017 [27].

4.2 Downloading the ISO and Preparing the Host System

Starting from a clean Ubuntu 12.04 LTS i386 system,[1] the first step is to install the required tools. The command

```
sudo apt-get install squashfs-tools genisoimage
```

can be run in the terminal to install both `squashfs-tools` and `mkisofs` at the same time [26]. Next, a copy of the installer needs to be downloaded from within the host system itself. The latest 12.04 LTS release can be downloaded

[1] It is easiest to start with the same username as the user that will be configured for the Live distribution, for reasons that will be seen later. For the instructions detailed here, that username is `npview`.

from `http://releases.ubuntu.com/`. Once the actual installer is present, a location for the temporary LiveCD compilation can be chosen and the ISO can be moved to that location. Assuming a folder called `livecdtmp` in the user's home directory is chosen as the folder where customizations are desired, the following commands can be performed [26]:

```
mkdir ~/livecdtmp
mv ubuntu-12.04-desktop-i386.iso ~/livecdtmp
cd ~/livecdtmp/
```

This command first creates the working directory, moves the ISO to that directory, and then enters the directory so that customization can be performed.[2]

At this point, the working directory only contains the ISO file that was downloaded. The ISO first needs to be mounted so that the files on it can be read. The filesystem also needs to be uncompressed so that changes can be made [26]:

```
mkdir mnt
sudo mount -o loop ubuntu-12.04-desktop-i386.iso mnt

mkdir extract-cd
sudo rsync --exclude=/casper/filesystem.squashfs -a mnt/ extract-cd

sudo unsquashfs mnt/casper/filesystem.squashfs
sudo mv squashfs-root edit
```

This first mounts the ISO to a directory called `mnt`, and then creates a directory called `extract-cd` to which all the LiveCD files are copied to. The LiveCD also contains the compressed filesystem, but this portion is omitted in copying. Finally the actual filesystem is uncompressed and moved to a folder called `edit`.

At this point, the system should be completely configured and ready for customization. Depending on what is needed, a few other small changes can be made, but the bulk of the preparation is done at this point.

[2]The name of the iso (`ubuntu-12.04-desktop-i386.iso`) *will* be different if the reader is working with a different distribution.

4.3 Final Preparation and Chroot

The last major preparation step is to `chroot`, or change root, into this system. But first, a few files need to be copied for Internet connectivity (to `wget` a file, or to `sudo apt-get install` anything, for example) [26]:

```
sudo cp /etc/resolv.conf edit/etc/
sudo cp /etc/hosts edit/etc/
```

These `resolv.conf` and `hosts` files will enable the user to download files from within the new environment, as they specify how to resolve Internet hostnames. These two files will need to be removed before the ISO is remastered, so that no networking hiccups occur when booting the LiveCD. If only a few packages are desired, however, they can simply be downloaded in the host system and then copied into the temporary filesystem without these Internet configuration files. This second method requires ensuring that file permissions and ownership on the newly copied files are set correctly.

Next, important directories for the temporary system are mounted, both from within and outside the new root environment [26]:

```
sudo mount --bind /dev/ edit/dev
sudo chroot edit
mount -t proc none /proc
mount -t sysfs none /sys
mount -t devpts none /dev/pts
```

This mounts the `/dev`, `/proc`, `/sys`, and `/dev/pts` directories as they should be, for any extensive customization within the new environment. If one is simply copying files into or editing existing files on the temporary filesystem, the `mount` commands are not required (only `sudo chroot edit`). Any installation using `apt-get` or compiling of programs will need these directories mounted, however. These directories will need to be unmounted before remastering the ISO, and especially before deleting the `edit` folder. If the `edit` folder is removed without performing the unmounts, these directories (and other parts of the host operating system) may become temporarily unstable until a reboot is performed.

The temporary filesystem has now been entered. Changes can now be made as if the temporary filesystem had been booted into directly.[3] Files can be

[3]The `chroot` environment can be exited by entering the simple `exit` command.

changed, utilities can be compiled, users can be set up, and programs can be installed using apt-get. At this point, most changes made to the system relate more directly to NetAPT and its dependencies. Although the reader is encouraged to see what specific changes are made to the system, they will most likely have different requirements and can fit these instructions to their own needs.

4.4 Making Changes within the Chroot Environment

The first tool that needs to be installed is `g++`. This will enable other tools to be compiled later on (specifically `xerces`). The command

```
apt-get install g++
```

will suffice. It is important to note that this and many other commands within the temporary environment do not require the `sudo` prefix. This is because the executed `chroot` command defaults to the `root` user.

4.4.1 Installing Ruby

Next, Ruby needs to be installed. Although this might normally easily be installed with a simple `apt-get install ruby1.9.3` command, most repositories that are usually available within a standard Ubuntu installation are not available here. If additional repositories are desired, one can add `http://archive.ubuntu.com/ubuntuprecise-universe` to `/etc/apt/sources.list`,[4] but this will only add slightly more standard packages. Not all expected packages will be available here, and some desired packages will have to be configured and installed by hand.

Before Ruby can be installed successfully, some dependencies need to be taken care of. First, `zlib` needs to be installed and is required for both Ruby and `libyaml`, which will be installed next. `Zlib` can be installed using a few quick commands [3, p. 98]:

```
wget http://www.zlib.net/zlib-1.2.8.tar.gz
tar -xvf zlib-1.2.8.tar.gz
```

[4] This repository URL will be different with versions of Ubuntu other than 12.04.

```
cd zlib-1.2.9/
./configure --prefix=/usr
make
make check
make install
```

After the installation is complete, a shared library needs to be moved to `/lib`. Because of this, the `.so` file in `/usr/lib` also needs to be recreated [3, p. 98]:

```
mv -v /usr/lib/libz.so.* /lib
ln -sfv ../../lib/libz.so.1.2.8 /usr/lib/libz.so
```

`Libyaml` is the next dependency required for correct installation of Ruby:

```
wget http://pyyaml.org/download/libyaml/yaml-0.1.4.tar.gz
tar -xvf yaml-0.1.4.tar.gz
cd yaml-0.1.4/
./configure
make
make install
```

Finally, Ruby can be installed [28]:

```
wget ftp://ftp.ruby-lang.org/pub/ruby/1.9/ruby-1.9.3-p392.tar.bz2
tar -xvf ruby-1.9.3-p392.tar.bz2
cd ruby-1.9.3-p392/
./configure --prefix=/usr --enable-shared
make
make install
```

In order to test that Ruby has been installed correctly, the command `gem --version` can be run. `Libyaml` was installed to help with errors about `psych` and Ruby not being installed correctly, but these also need `zlib` as well.

Next, some Ruby gems need to be installed:[5]

[5]For `sqlite3`, `rails`, and `netaddr`, internet connectivity is needed. For `antfarm-uiuc-0.4.0`, the gem needs to be copied into the directory where this command is executed.

```
gem install --remote sqlite3
gem install rails
gem install netaddr
gem install antfarm-uiuc-0.4.0.gem
```

At this point, several of NetAPT's dependencies have been successfully installed.

4.4.2 Setting Up the User

Next, a new user needs to be added and configured. For the NetAPT project, this is because it is desirable to have a user pre-configured when a customer uses the ISO. This way, that user's settings, desktop background, navigation bar, shortcuts, and home folder can already be configured. Remember, ANTFARM also requires the `.antfarm` directory in the user's home folder for proper runtime configuration. The user can be added with a few simple commands:

```
useradd npview
passwd npview

adduser npview sudo
adduser npview adm
```

This has created a new user called `npview`.[6] The next command is to set that user's password.[7] Finally, the user is added to the `sudo` group, for administrator permissions, and to the `adm` group, so that it has access to the `/var/log` folder. Finally the default shell of the `npview` user can be set to bash by editing `/etc/passwd`. The location after the last colon (`:/bin/bash`) controls the user's default shell.[8]

[6] NetAPT is in the process of changing its name to NP-View. Any mentions to NP-View are for forward compatibility.

[7] The password is not created simply by entering the `passwd` command, but in the dialogs that follow. The user will be asked to enter the new user's password.

[8] Be careful not to make any other edits to this file as they may drastically change the behavior of the system.

4.4.3 Configuration As the New User

The temporary filesystem now has several additional programs installed, and a new user as well. It is time to make some changes as that new user. A command is issued to switch to the new `npview` user:

```
su - npview
```

Once `npview`'s password is entered, all subsequent changes will be made as the `npview` user.[9]

It is first desired to install and configure just a few more dependencies. These programs need to be installed as the actual user that uses them, rather than the root user, due to their location in the filesystem. The first program to install is `openssl`, which can be installed easily:

```
cd ~
wget http://www.openssl.org/source/openssl-1.0.1c.tar.gz
tar -xvf openssl-1.0.1c.tar.gz
cd openssl-1.0.1c/
./config
make
sudo make install
cd ..
rm openssl-1.0.1c.tar.gz
```

Notice that `openssl` has been installed in the user's home folder. It is important that the original directory remain there because it is needed to run NetAPT. The programs that were installed earlier did not need to be installed to the user's home folder, and were installed either to the default system location or a to specified prefix.

`Xerces-C` also needs to be installed:

```
wget http://apache.osuosl.org//xerces/c/3/sources/xerces-c-3.1.1.tar.gz
tar -xvf xerces-c-3.1.1.tar.gz
cd xerces-c-3.1.1/
./configure
make
sudo make install
```

[9] To stop making commands as the new user, use the `exit` command.

```
cd ..
rm xerces-c-3.1.1.tar.gz
```

Now that `openssl` and `xerces-c` have been installed, `ANTFARM` can be configured [1]:[10]

```
sudo antfarm db --initialize
sudo antfarm db --migrate
sudo chown -R npview:npview .antfarm
```

This first creates the `.antfarm` folder inside the user's home folder. Next, the `migrate` command initializes the `ANTFARM` Sqlite database. Finally, the third command makes sure that the correct user owns the `.antfarm` directory.[11] At this point, any desired scripts, including the `ANTFARM` scripts that NetAPT uses, should be copied into the `~/.antfarm/scripts` directory so that they can be used by NetAPT [1].

NetAPT and Vizir have to be compiled. NetAPT can be compiled using the `make` command along with the makefile downloaded with the repository, shown in Appendix B. The Vizir GUI is built using `Apache Ant`, and can be compiled with a simple `ant` command entered within the Vizir build directory.

Next, the APT and Vizir folders can be copied into the user's home directory. As these files are copied in, any unnecessary subdirectories (such as `3rdparty`, `doc`, and `test`) can be omitted. Many of the directories have no impact on the function of the tool, and a few directories (`src`, in particular) should not be accessible by potential users of the tool. Once these directories have been made, a static link needs to be created to fix an inconsistency in how netapt is run:[12]

```
ln -s /home/npview/Vizir/keysets /home/keysets
```

Finally, the most recent version of the Java Standard Edition Development Kit (Java SE Development Kit, or JDK) can be downloaded and extracted

[10] Before this step, `/usr/lib/ruby/gems/1.9.1gems/antfarm/lib/init/initializer.rb` might need to be modified. The line that says `require Logger` should be `require logger`.

[11] Since `ANTFARM` has been created and installed using `sudo` commands, permissions may not be correctly set.

[12] This static link needs to be placed in the *parent* directory of the folder where shell command is executed, or the tool will throw an error about not being able to access the *keysets* directory.

to the `/home/npview` directory. NetAPT needs Java to be able to display the GUI, and this does not come standard in the installer.[13]

At this point, it is fine to `exit` out of the `su` environment, as no more customizations need to be made as the `npview` user.

4.5 Setting Up the User Environment

The distribution is almost configured exactly how it should be. All the necessary programs are installed, and just a few more small changes have to be made. First, NetAPT needs several environment variables to be set when it runs. In order for the program to run correctly, these should be set as system-wide environment variables in `/etc/environment`. These variables need to specify the locations of several of the tools that were just installed:[14]

```
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:\
       /usr/bin:/sbin:/bin:/home/npview/jdk1.7.0_21/bin"
NETAPT_ROOT=/home/npview/APT
VIZIR_ROOT=/home/npview/Vizir
OPENSSL_ROOT=/home/npview/openssl-1.0.1c
XERCESCROOT=/home/npview/xerces-c-3.1.1
JAVA_HOME=/home/npview/jdk1.7.0_21
```

The `exit` command can be entered once again to exit out of the `chroot` environment. Before exiting, any temporary or history files that may have been created within the `chroot` need to be deleted. If the networking files were copied in earlier, those need to be removed as well. If any software was installed, the `machine-id` file needs to be removed also. Finally, the system directories that were mounted earlier need to be unmounted [26]:

```
rm -rf edit/tmp/* edit/root/.bash_history edit/npview/.bash_history
rm edit/etc/hosts edit/etc/resolv.conf
rm edit/var/lib/dbus/machine-id
```

[13] It is extremely important to get the *most recent* version of the JDK, as bugs in Java are exploited on a routine basis.

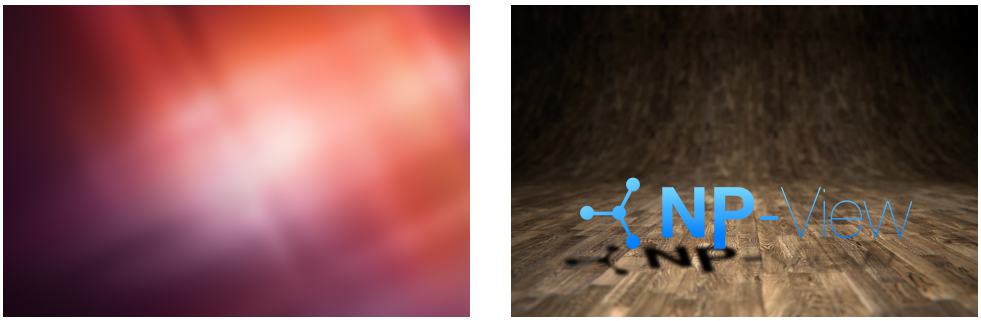
[14] These commands will need to be amended if different versions of the aforementioned utilities have been downloaded.

```
umount /proc || umount -lf /proc
umount /sys
umount /dev/pts

exit

sudo umount edit/dev
```

Next, it is important to configure the actual user experience that the user will have. The default background can be customized. Any `1920x1280` PNG can be copied into `~/livecdtmp/edit/usr/snare/backgrounds/warty-final-ubuntu.png`[15] This sort of customization is a big visual change, and can change the background from something that's a little boring to a much more branded image, as seen in Figure 4.1.



(a) Ubuntu stock background (b) Customized background

Figure 4.1: Customizing the Desktop Background

If desired, the branding of the whole system can be customized. The logo for the operating system itself, `ubuntu_logo.png`, in `~/livecdtmp/edit/lib/plymouth` can be replaced with any other `217x58` pixel PNG. For even deeper customization of the startup and shutdown screens, the files in `~/livecdtmp/edit/lib/plymouth/themes/ubuntu-logo` can be replaced with counterparts of the exact same size.

Finally, the desktop itself can also be customized. It would be desirable to have a shortcut on the Desktop and the Unity Bar to be able to click and run NetAPT. This can be done with a tool called `gnome-panel`:

[15] Once again, if a different distribution is being used, the name of this file may be different.

```
sudo apt-get install gnome-panel
gnome-desktop-item-edit --create-new ~/Desktop
```

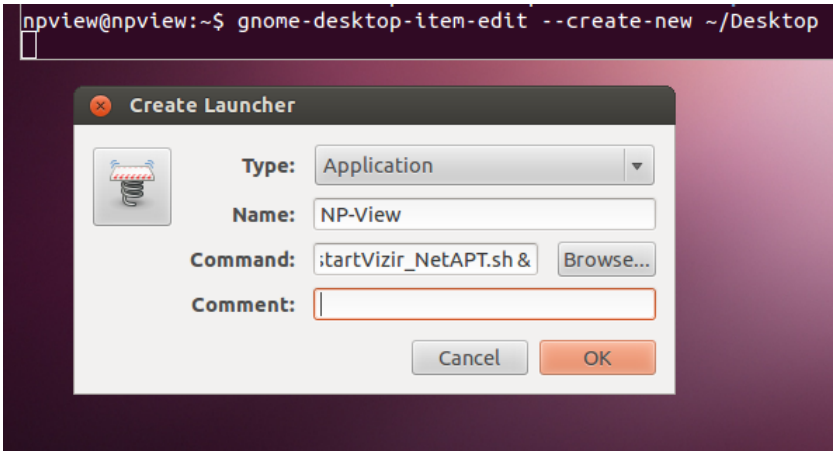


Figure 4.2: Creating a Custom Launcher

This will create an Application Launcher shortcut on the Desktop. This shortcut can be edited so that it shows whatever name and icon are desired, but the command line in the applicable box should be set to `/home/npview/Vizir/startVizir_NetAPT.sh &` (Figure 4.2). This will ensure that the correct script is run when the Launcher is clicked. This Launcher can be copied onto the Unity Bar for easy access as well. Finally, the Unity Bar can be edited so that it shows only those applications that might be utilized in the system (Figure 4.3) [29].

Note that the Desktop and Unity Bar are currently being configured in the `host` environment. This is acceptable, since they will be copied into the new environment and cannot be configured within a `chroot`. They depend on graphical utilities that are specified by certain configuration files; it is easiest to arrange them graphically as desired, and then copy the configuration files in later.

Once the Desktop and Unity Bar are configured as desired, they can be copied into the folders in the temporary filesystem so that they will be applied to the new user upon installation or when the LiveCD is booted:

```
mkdir -p ~/livecdtmp/edit/home/npview/.config/dconf
cp -r ~/.config/dconf/user \
       ~/livecdtmp/edit/home/npview/.config/dconf
cp -r ~/Desktop ~/livecdtmp/edit/home/npview/
```

Figure 4.3: Ubuntu's Unity Bar

These folders are normally created when the user is initialized, but if the folders already exist when this happens, they will not be overwritten. This provides the opportunity to exactly configure certain aspects of the user's experience. The first `user` configuration file in the `dconf` folder specifies the configuration for the Unity Bar. When this file is copied in, the new user's Unity Bar will look exactly the same. The second file that is copied is the Desktop itself, which contains the Application Launcher shortcut that was just created.

4.6 Final Changes

Finally, the User ID (*UID*) and Group ID (*GID*) for the live user need to be changed. In Linux, all IDs below 1000 are reserved for system use, while those 1000 and above are for actual users on the system. When a new user

is created, that user gets a User ID, but a new group is also created. That group typically has the same group name as the user's username (in this case, `npview`), and it has to get an ID as well. Linux assigns new UIDs and GIDs based on the next number available, starting from 1000. So, for example, if both the UIDs and GIDs 1000-1026 are taken, the system would assign the next created user to a UID (and GID) of 1027. Note that UIDs and GIDs do not always have to match up, so Linux could assign a new user a different GID than its UID, if different numbers are available. Often, these numbers will match up, however, because there are not typically that many users and groups on a system.

Currently, the UID and GID for the new `npview` user are currently set to `1000`. Linux correctly assigned the first number above 999 to the new user. When the system boots, however, there will be some incompatibilities. When the LiveCD is booted, the system is currently set to create a new live user with a UID of 1000. Since that UID is already taken on the system, it does not boot. A simple solution would be to change the UID of the user to 2000 with a command like `sudo usermod -u 2000 npview`. Although this would accomplish the job, it is a bit of a hack: when the new system is booted up, everything works correctly, although there may be messages about an `authentication failure`. `Authentication failure` messages flying by as the system boots up are certainly not ideal. Critically, if the new system is installed, the user that has been explicitly set up (`npview`) will show up, but any new users that were configured during the installation process fail to display as a login option later.

This sort of issue can be circumvented in a better way. While IDs from zero to 999 are typically reserved for system use, nothing will prevent setting a user's ID to less than that. Those users can also be used just as they normally would, with one small change, which will be explained shortly. The UID and GID for the new `npview` user can be set to 991 by entering a few quick commands:[16]

```
sudo usermod -u 991 npview
sudo groupmod -g 991 npview
```

These two commands respectively change the UID of the `npview` user to

[16]The group name for the user is almost never different than the username, but if it is, the second command will have to include the group name rather than the username.

991, and the GID of the npview group to 991. Now, the npview user can be utilized just like any other user. When the installation of the LiveCD happens, there will be no authentication failures and any users created as a part of the installation process will show up as well. Two small changes need to be made, however. First, when Ubuntu starts up, the GUI only shows users that have a UID of 1000 or more. Since the npview user's UID is set to 991, it will not show up in the login screen. Ubuntu also needs to use the npview user as the live user when the LivdCD is booted. Both of these things can be changed fairly easily.

The problem of npview not showing up on the login screen can be fixed by changing one simple file. The login.defs file in ~/livecdtmp/edit/etc has several variables that define things related to login. If the UID_MIN variable on line 167 is changed to 990, the npview user will now show up in the login screen! Configuring the live user is just a little bit more work.

For this, the initrd has to be customized. Although only one file needs to be changed, the whole thing has to be unpacked, modified, and repackaged. From within the ~/livecdtmp folder, execute the following commands [30]:

```
mkdir initrd-tmp
cd initrd-tmp
lzma -dc -S .lz /extract-cd/casper/initrd.lz | cpio -id
```

These will unpack the current initrd to a new folder called initrd-tmp. Once the files have been extracted, the file in /etc/casper.conf has to be edited. Change it to look something like this:[17]

```
# This file should go in /etc/casper.conf
# Supported variables are:
# USERNAME, USERFULLNAME, HOST, BUILD_SYSTEM, FLAVOUR

export USERNAME="npview"
export USERFULLNAME="NP-View Live session user"
export HOST="npview"
export BUILD_SYSTEM="NP-View"
```

[17] If the FLAVOUR setting is not set, the other live user settings will not be changed either. This variable must be set.

```
# USERNAME and HOSTNAME as specified above won't be honoured
# and will be set to flavour string acquired at boot time,
# unless you set FLAVOUR to any non-empty string.

export FLAVOUR="NP-View"
```

This changes the live user of the LiveCD to the `npview` user that was set up before. Once this has been done, the `initrd` can be repacked [30]:

```
find . | cpio --quiet --dereference -o -H newc | \
       lzma -7 > ../new-initrd.lz
cd ../
cp new-initrd.lz extract-cd/casper/initrd.lz
```

The new `initrd` is now copied into where the old one was.[18]

4.7 Making the ISO

Once the `initrd` has been customized, the system has been completely configured and is ready to be packaged! Make sure that all temporary, history, networking, and `machine-id` files have been removed, and that all mounted system directories have been unmounted, as mentioned before. For the rest of these commands, the `su` environment can be reentered. The first task is to regenerate the manifest file [26]:

```
chmod +w extract-cd/casper/filesystem.manifest
chroot edit dpkg-query -W --showformat='${Package} \
       ${Version}\n' > extract-cd/casper/filesystem.manifest
cp extract-cd/casper/filesystem.manifest \
       extract-cd/casper/filesystem.manifest-desktop
sed -i '/ubiquity/d' \
       extract-cd/casper/filesystem.manifest-desktop
sed -i '/casper/d' \
       extract-cd/casper/filesystem.manifest-desktop
```

[18] When modifications to the `initrd` have been made, the system will not boot with VMWare. To use the system on VMWare, it is best to simply set the UID to 2000.

The *manifest* is a file present in most distributions that specifies exactly what packages are present in the distribution. First, this file is made writable, and then the manifest is written to the file. It is copied to the correct location in the filesystem and a few small changes are made.

Next, the filesystem needs to be compressed. A compression format called *squashfs* is used to help greatly reduce the size of the files and folders on the system.[19] The filesystem can be compressed with the commands [26]:

```
rm extract-cd/casper/filesystem.squashfs
mksquashfs edit extract-cd/casper/filesystem.squashfs
```

The command-line switch `-b 1048576` can alternatively be added to the end of the previous command. This increases the block size, which provides slightly higher compression at the cost of just a little more time to perform that compression.

Next, the `filesystem.size` and image name need to be updated [26]:

```
printf $(du -sx --block-size=1 edit | \
       cut -f1) > extract-cd/casper/filesystem.size
nano extract-cd/README.diskdefines
```

The `filesystem.size` file helps determine what size the entire compressed filesystem is, so that it can be used by lower-level processes. Within the `README.diskdefines` file, the `DISKNAME` definition is probably the only one that needs to be changed.

Next, the *md5sums* have to be recalculated to ensure that the system recognizes that any changes that have been made were intentional [26]:

```
cd extract-cd
rm md5sum.txt
find -type f -print0 | xargs -0 md5sum | \
       grep -v isolinux/boot.cat | tee md5sum.txt
```

All necessary ISO files have now been recalculated. Issue one final command to generate the ISO itself [26]:

```
mkisofs -D -r -V "$IMAGE_NAME" -cache-inodes -J -l -b \
```

[19]The kernel needs to be compiled with `squashfs` support for this to work. In the Ubuntu installer, this has already been taken care of.

```
isolinux/isolinux.bin -c isolinux/boot.cat \
-no-emul-boot -boot-load-size 4 -boot-info-table \
-o ../NP-View-[tool-version]-[iso-version]-i386.iso .
```

The ISO will be created in `~/livecdtmp/`. After this is complete, the `su` environment can be `exit`ed.

CHAPTER 5

CONCLUSION AND FUTURE WORK

NetAPT is employed by enterprise and utility customers to help validate their network security policy. It is a useful and powerful tool, but relies on the help of some extremely stubborn dependencies. Configuring and installing these dependencies often takes much more time than learning how to use the tool itself. Although installers are currently available for Windows and Mac, they only partially solve the configuration dilemma and fail to function as well as they should. This research set out to create an easier way to use NetAPT by taking some of the hard configuration and setup out of the hands of the user.

LFS was first used, which allows a system to be built completely from scratch, the advantage of which is inclusion of only those absolutely necessary parts of the operating system. The minimal and bloat-free system took a long time to build, but the result was a distribution that lacked real practicality and purpose, and failed to function as intended. The system did not have the bootscripts to handle mounting a ramdisk, and ended up being primarily read-only, which meant that the GUI could not be run. Although the distribution was simple at best, at 2 GB it was too large and finicky to be useful or satisfy the initial requirements.

An adjustment of tack produced a viable alternative. The Ubuntu LiveCD ships as a 700 MB ISO, ready to run as a LiveCD or installer. The ISO itself can be mounted and edited for deep customization, allowing for creation of a new distribution that well fits the intended purpose. NetAPT and its myriad of dependencies were installed and configured along with a new user that could take advantage of all the customization. The background, startup, and shutdown screens were changed, and desktop and Unity Bar shortcuts for the application were created. In the end, this provided a fairly branded and consistent product that could easily run NetAPT without configuration.

The system was designed to meet three goals: it needed to have a small

footprint, the ability to run as a LiveCD, and a small amount of bloat. The final distribution weighed in at just over 970 MB, which is not only impressively small, but also well within the requirement. Users will be able to download updates to the tool in approximately 18 minutes on an average connection. Once downloaded, the tool is ready to be used, taking up far less time than configuration and installation had before.

Secondly, the tool runs quite well as a LiveCD. Customers can start using the tool immediately after it is obtained, without the need for installation. This is helpful for using the tool on a computer without affecting the existing installed OS. The ISO is flexible and can be burned to a bootable DVD or copied to a LiveUSB. If needed, the distribution can still be installed natively to a computer. The installation is the exact same setup as the LiveCD, so everything runs as expected. The system can be virtualized on a number of different platforms as well. This compatibility with virtualization software allows enterprise users to set up and properly sandbox the tool so that security can be maintained.

Finally, the system is built from a traditional Ubuntu installer, so it ships out-of-the-box with all of the programs that come standard on Ubuntu. Instead of adding bloat, these extra utilities make the system feature-complete and usable. If a user wants to make notes on a spreadsheet, look up a configuration question online, or copy the tool's results to a USB drive, these things are all possible, and would not be without a more minimal system. As a matter of fact, the compression used on the distribution makes it much more space efficient than the completely minimal and bloat-free system that was first attempted.

Further research and development could extend the functionality of this product. First, the created operating system is a 32-bit OS. While many computers (and virtualization solutions) only support 32 bits, it would be helpful to produce a 64-bit version as well. Those users that have 64-bit processors could take full advantage of their architecture to get the most power out of the tool. Creating such a distribution would be straightforward; the instructions provided here could easily be extended to a 64-bit OS. The host architecture needs to match the architecture of the intended distribution. This means the chosen host operating system would have to be a 64-bit OS, and the tool would need to be recompiled for 64 bits, but most of the other configuration instructions would be exactly the same.

Second, it would be helpful to find a solution that would allow the created user to be designated as the live user, while still maintaining compatibility with all virtualization solutions. In order to designate the new user as the live user, the init ramdisk has to be customized. This change means that one of the most popular virtualization products, VMWare, cannot run the ISO. Currently, this means that two separate ISOs can be created, with separate solutions for each. If there were a solution that worked consistently across all platforms, this would be ideal.

Third, the branding of the created distribution could be improved. While certain customizations such as the installer text, startup and shutdown screens, and background image currently brand the OS fairly well, this branding customization could be taken to the next level. If every aspect of the new operating system were branded, from the installation process to the official title of the running operating system, this would immerse the user in a professional product. Distributions such as Edubuntu and Linux Mint already perform this sort of deep customization on a routine basis. Although these distributions prove that extensive branding as a concept is possible, it may require more resources than the current team has to offer.

Finally, it would be helpful to figure out a solution that incorporated automation, or at least a scripted build. This would be useful when new versions of the tool or new LTS releases of Ubuntu were publicized. Currently, new versions of the tool can be updated within the build environment, but then a new ISO has to be packaged and created, which is not a quick process. A transition to a new version of Ubuntu is much less straightforward: the whole process has to be repeated from the beginning to transition to a newer host OS. There might be a mechanism to auto-update the OS from within the build environment, but this has not been explored. If automation were incorporated, these tasks would be made much less difficult, as a simple command could be run to perform all the compilation.

Overall, the created distribution not only well fits the intended purpose, but is easy to use, feature-complete, and flexible. It takes the burden of configuration out of the hands of the user, and minimizes the learning curve and effort needed to start using the tool. With the hassle of configuration out of the way, users are free to start using the tool as it was intended: to help make their networks more efficient, reliable, and secure.

APPENDIX A

A CLOSER LOOK AT NETAPT

NetAPT works in a four-step process. First, it reads in a set of firewall configuration files, and uses them to draw a map that visualizes the network. Next, it parses the files to determine what specific rules have been configured and their impact on the network. After a network security policy has been specified, it looks at the complex ways in which the rules of these firewalls interact, and finally draws an interactive representation of all traffic on the network. This representation makes it much easier for a network administrator to determine whether everything has been configured correctly.

The tool first starts by reading the specified firewall configuration files. From this file, it can automatically determine what firewall vendor is being used. NetAPT was first developed with support for Cisco firewalls. As customers began to use the tool, they started to ask for support of more firewalls. Rudimentary support of Checkpoint and Sonicwall firewalls was later added. As Cisco continued to iterate on its firewall standard with versions 8.3 and 8.4, NetAPT gained support for those as well. Once the tool knows what the syntax should look like, it begins to parse through the files and collect an internal representation of the topology of the entire network. This task is farmed out to several different Ruby scripts that are used for parsing the exact syntax of the configuration file. From these files, the tool is able to accurately and intelligently determine which networks connect together. Once all the information about topology has been collected, this internal representation is fed into the main GUI, where a tool called ANTFARM is used to display all the network nodes in a readable graph, shown in Figure A.1.

The user now has an accurate representation of the network topology. The second task is to load in the rules from the firewall configuration files. This is similar to the first task, although instead of concentrating on specific nodes or endpoints in the network, the parser focuses on the specific rules that define

what nodes are able to communicate with what other nodes. Again, these rules can be quite complicated. For example, a rule can not only define which computers can talk to each other, but can also specify in which direction those communications happen, over which ports, and what protocols are allowed. Once again, the tool uses Ruby scripts to do the heavy lifting, and imports a list of each and every rule that appears in the selected group of firewall configuration files.

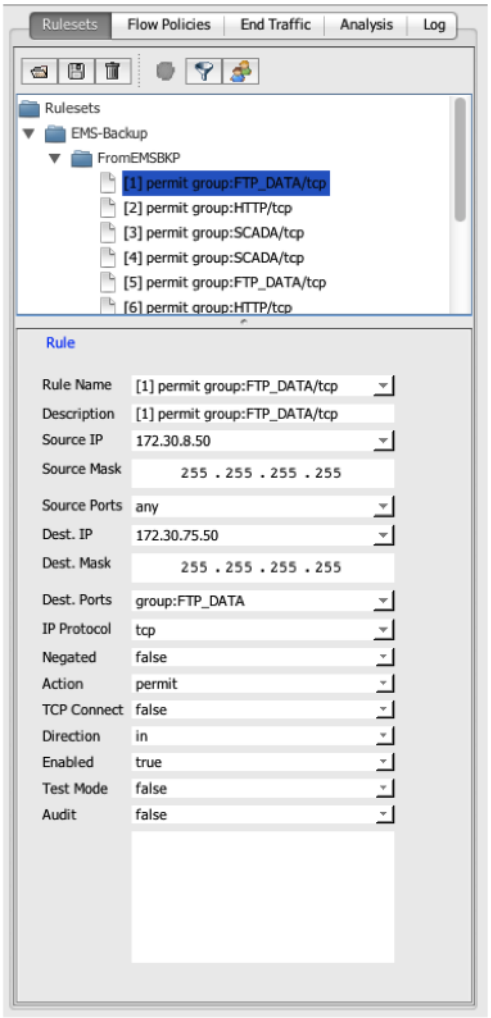


Figure A.1: NetAPT's Rule Pane

The user can see a map of the network topology (Figure 1.1 on page 3), and a corresponding set of rules (Figure A.1). The third step is to specify the desired policy. Once the tool knows about the security policy of the network, it can evaluate all the firewall configuration rules to determine if any of them violate the specified policy. Although the tool was made to

handle many different kinds of security policy, the main use case is to specify a global `deny any any` policy, which means that each and every connection is denied.[1] The tool then looks for violations of that policy, which shows the user every single flow through the entire network. In short, it shows every single possible connection from one node to another within the network.

Once the policy has been specified, the tool can perform the final step: analysis. This is the step where NetAPT actually determines what firewall configuration rules violate the specified policy. Each and every path through the network has to be checked against every rule of every firewall configuration file. Since firewalls can have thousands of rules, this is easily the longest step in the process. For a modest network, this might take an hour or so to complete, but can definitely take much longer for a larger network with more firewalls. Even though this step takes quite a long time, the results are well worth it.

[1] The first `any` specifies all IP addresses, while the second enumerates all ports on those IPs.

APPENDIX B

NETAPT MAKEFILE

```
# $Id: Makefile.linux, v 1.3.3
# 2013-05-23 02:36:06 zachyordy

# Make sure to set NETAPT_ROOT and
# XERCES_PATH to their actual value
NETAPT_ROOT ?= /root/APT
XERCESCROOT ?= /root/xerces-c-3.1.1
OPENSSL_ROOT ?= /root/openssl-1.0.1c

OPENSSL_INC = -I "${OPENSSL_ROOT}/include"
OPENSSL_LIB = -L "${OPENSSL_ROOT}"

XERCESC_INC = -I "${XERCESCROOT}/src"
XERCESC_LIB = -L "${XERCESCROOT}/src/.libs"

CC=g++
CFLAGS=-c $(XERCESC_INC) $(OPENSSL_INC) -I \
       "$(NETAPT_ROOT)"/src/include -g -Wall
LDFLAGS=$(XERCESC_LIB) $(OPENSSL_LIB) -ldl  \
       -static-libgcc -static-libstdc++
INSTALL=../bin

# Server source and executable
SERVER_SRC=rulegraph.cpp SAX2APTHandler.cpp  \
       AnalysisEngineServer.cpp
SERVER_OBJS=$(SERVER_SRC:.cpp=.o)
SERVER=NetAPTServer
```

```
all: $(SERVER_SRC) $(SERVER)
rm -f *.o

$(SERVER): $(SERVER_OBJS)
$(CC) $(LDFLAGS) $(SERVER_OBJS) -lssl  \
       -lcrypto -lxerces-c -lpthread -o $@
rm -f *.o
mv  $(SERVER) $(INSTALL)

.cpp.o:
$(CC) $(CFLAGS) $< -o $@

clean:
rm -f *.o $(INSTALL)/$(SERVER)
```

REFERENCES

[1] B. T. Richardson, "ANTFARM Application Documentation, Version 0.4.0," Oct. 2008. [Online]. Available: http://antfarm.rubyforge.org/

[2] D. Belson et al., "Executive Summary," *The State of the Internet*, vol. 5, no. 4, Jan. 2013.

[3] G. Beekmans, M. Burgess, and B. Dubbs, "Linux From Scratch: Version 7.3," Mar. 2013. [Online]. Available: http://www.linuxfromscratch.org/lfs/downloads/stable/LFS-BOOK-7.3.pdf

[4] T. Anderson, "The Case for Application-Specific Operating Systems," in *Third Workshop on Workstation Operating Systems*, Apr. 1992, pp. 92–94.

[5] R. Kumar, S. K. Narayanasamy, E. Perelman, and P. Gupta, "Survey on Application Specific Operating Systems," 1995, unpublished. [Online]. Available: http://cseweb.ucsd.edu/classes/fa01/cse221/projects/group4.ps

[6] L. F. Friedrich, J. Stankovic, M. Humphrey, M. Marley, and J. Haskins Jr., "A Survey of Configurable, Component-Based Operating Systems for Embedded Applications," *Micro, IEEE*, vol. 21, no. 3, pp. 54–68, June 2001.

[7] R. Budihal, "Emerging trends in embedded systems and applications," July 2010. [Online]. Available: http://www.eetimes.com/discussion/other/4204667/Emerging-trends-in-embedded-systems-and-applications

[8] C.-T. Lee, J.-M. Lin, Z.-W. Hong, and W.-T. Lee, "An Application-Oriented Linux Kernel Customization for Embedded Systems," *Journal of Information Science and Engineering*, vol. 20, pp. 1093–1107, June 2004.

[9] J.-C. Tournier, "A Survey of Configurable Operating Systems," unpublished. Accessed: May 2013. [Online]. Available: http://www.cs.unm.edu/~treport/tr/05-12/Tournier.pdf

[10] L. Youseff, "Application Specific Operating Systems," unpublished. Accessed: May 2013. [Online]. Available: http://www.cs.ucsb.edu/~lyouseff/pdfs/MAE_Document.pdf

[11] J. Dias, S. Tavares, A. Carvas, and P. S. Silva, "Open Source Operating System for Students: EOS Project," in *Workshop on Open Source and Design of Communication*, vol. 20, Lisbon, Portugal, June 2012, pp. 79–83.

[12] Canonical, Ltd., "The World's Most Popular Free OS — Ubuntu," Accessed: Nov 2012. [Online]. Available: http://www.ubuntu.com/

[13] E. Worden, "Installation/MinimalCD," Oct. 2008. [Online]. Available: https://help.ubuntu.com/community/Installation/MinimalCD

[14] F. Balliano, "Ubuntu Mini Remix - The tiny Ubuntu you can build on!" 2011. [Online]. Available: http://www.ubuntu-mini-remix.org/

[15] Canonical, Ltd., "Lubuntu — lightweight, fast, easier," Accessed: Nov 2012. [Online]. Available: http://www.lubuntu.net/

[16] J. Hoogland et al., "Bodhi Linux," 2013. [Online]. Available: http://www.bodhilinux.com

[17] "SlimPup," Accessed: Nov 2012. [Online]. Available: http://slimpuplinux.sourceforge.net

[18] J. Andrews, R. Lindsay, et al., "DSL Information," 2012. [Online]. Available: http://www.damnsmalllinux.org

[19] A. Coopersmith et al., "Home - X.org Wiki," Apr. 2013. [Online]. Available: http://www.x.org

[20] R. McMurchy, B. Dubbs, et al., "Introduction to Xorg-7.7," May 2013. [Online]. Available: http://www.linuxfromscratch.org/blfs/view/svn/x/xorg7.html

[21] R. McMurchy, B. Dubbs, et al., "Xorg Applications," June 2013. [Online]. Available: http://www.linuxfromscratch.org/blfs/view/svn/x/x7app.html

[22] R. McMurchy, B. Dubbs, et al., "Fluxbox-1.3.5," June 2013. [Online]. Available: http://www.linuxfromscratch.org/blfs/view/svn/x/fluxbox.html

[23] "Volume and File Structure of CDROM for Information Interchange," Dec. 1987. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-119.pdf

[24] H. P. Anvin, "ISOLINUX - Syslinux Wiki," June 2013. [Online]. Available: http://www.syslinux.org/wiki/index.php/ISOLINUX

[25] H. P. Anvin, "Osolinux.cfg - Syslinux Wiki," Sep. 2012. [Online]. Available: http://www.syslinux.org/wiki/index.php/Isolinux.cfg

[26] M. East, P. Bull, J. Bicha, and B. Kerensa, "LiveCDCustomizationFromScratch," Apr. 2013. [Online]. Available: https://help.ubuntu.com/community/LiveCDCustomizationFromScratch

[27] R. Ancell, "Lts," Mar. 2013. [Online]. Available: https://wiki.ubuntu.com/LTS

[28] R. McMurchy, B. Dubbs, et al., "Ruby-1.9.3-p392," Apr. 2013. [Online]. Available: http://www.linuxfromscratch.org/blfs/view/svn/general/ruby.html

[29] "How can I edit/create new launcher items in Unity by hand?" Accessed: May 2013. [Online]. Available: http://askubuntu.com/questions/13758/how-can-i-edit-create-new-launcher-items-in-unity-by-hand

[30] M. Ledbetter, "CustomizeLiveInitrd," Mar. 2012. [Online]. Available: https://wiki.ubuntu.com/CustomizeLiveInitrd